**UNIVERSIDAD AUTÓNOMA DE MADRID**
**ESCUELA POLITÉCNICA SUPERIOR**


**Degree as**

# DEGREE WORK

## Functional Programming Paradigm for Machine Learning Algorithms in Data Mining

**Author: Miguel Fernández de Alarcón Gervás**
**Advisor: Francisco Saiz López**

**junio 2019**

**Miguel Fernández de Alarcón Gervás**
***Functional Programming Paradigm for Machine Learning Algorithms in Data Mining***

**Miguel Fernández de Alarcón Gervás**
C\ Francisco Tomás y Valiente Nº 11

PRINTED IN SPAIN

# Agradecimientos

Quiero agradecerles el trabajo que han realizado conmigo y el tiempo que me han dedicado mis tutores Francisco Saiz y David Camacho. A mi familia por su apoyo, a mis compañeros de clase por hacer las jornadas en la biblioteca más pasables. A Roberto por ayudarme cuano estaba empezando, a Merel por sus conversaciones en los momentos en los que necesitaba desconectar, a Suso por sus descansos de 5 minutos que se alargaban otros 5 o 40 minutos.

Quiero agradecerle especialmente a Emily y Lisa, por haberme ayudado en la corrección de erratas y expresiones en el documento y a todos los que han estado conmigo en este proceso.

# Resumen

La minería de datos y el aprendizaje automático han visto un auge en las últimas décadas, con ordenadores cada vez más potentes y, principalmente, debido a la cantidad de datos a los que se puede acceder en la actualidad. Esta cantidad de datos no hace más que aumentar, y está facilitando la llegada de tecnologías que permiten su procesado para obtener diversos beneficios, desde la conducción automática hasta mejoras en el diagnóstico de enfermedades.

El lenguaje por excelencia en la minería de datos es Python, por su versatilidad y la cantidad de librerías disponibles para tratar los datos, como aplicar algoritmos para obtener resultados, incluyendo herramientas para mostrar gráficamente los resultados. El objetivo de este proyecto es comparar un paradigma de programación funcional frente a Python. Como lenguaje de programación funcional hemos escogido Haskell, por ser un lenguaje de programación puramente funcional.

El proyecto pretende comparar ambos lenguajes a partir de resultados de dos algoritmos de clasificación frecuentemente utilizados: Random Forest (RF) y Redes Neuronales Convolucionales (CNN). Los resultados obtenidos se inclinan muy a favor de Python para Random Forests y Redes Neuronales Densas (FFNN), con tiempos hasta diez veces inferiores y consumos de memoria de un tercio al usado con Haskell. La diferencia en tiempos de ejecución es considerablemente menor para las convolucionales, tardando Haskell aproximadamente 1.5 veces más que Python, aunque el consumo de memoria sigue difiriendo en un factor de 2-3. Por tanto, concluimos que Haskell no es lo suficientemente eficiente para suplantar a Python en el campo del aprendizaje automático. Sin embargo, no se descarta como solución para otros problemas por las ventajas que ofrece como, por ejemplo, a la hora de paralelizar la ejecución del código o a la seguridad del mismo.

# Palabras clave

Haskell, Python, funcional, comparación de rendimientos, escalabilidad, mineria de datos, aprendizaje automático

# Abstract

Data Mining and Machine Learning have experienced a significant rise in the past few decades, due to the increasing computational power, as well as to the quantity and availability of data which we currently have access to. This amount of data is always growing and facilitates the arrival of technologies to process them in order to obtain some benefit out of it, from autonomous driving to enhancements in the diagnosis and treatment of diseases.

The programming language most widely used in data mining is Python, primarily because of its versatility and the number of libraries that are available for data processing, the different machine learning algorithms and tools to provide visual analysis of datasets and results. The goal of this project is to compare a functional programming paradigm to Python. Our choice of functional programming language is Haskell, owing to its purely functional nature.

The study aims to compare the two languages based on the results of their performance when using two common classification algorithms: Random Forest (RF) and Convolutional Neural Networks (CNNs). The experimental results lean towards Python being the better solution, with running times of up to ten times faster and a third of the memory consumption in both Random Forests and Feed Forward Neural Networks (FFNN). The differences in performance were not as significant for CNNs, with Haskell needing around 1.5 times the amount of time Python requires, although memory is still around 2-3 times as much. Thus, we can conclude that Python is still the language to beat in the field of machine learning. Despite this, Haskell can still be used as a solution to other problems that can make use of its best qualities, which include its strong typing, safety because of its lack of side effects and its ease to safely parallelize code.

# Keywords

Haskell, Python, functional, performance comparison, scalability, data mining, machine learning

# Table of Contents

# Lists

## List of algorithms



## List of codes



## List of equations



## List of figures

# List of tables

# 1

# Introduction

## 1.1. Project Motivation

There is no programming language or paradigm which is better than the rest. Each one has its strong and weak points, so the question should be: Is this particular language the best solution for this specific task?

The main goal of this project is to view if there is a functional alternative to the most widely used programming language in the data miningfield of machine learning: *Python*. Even though Python can be considered a functional language, it is a hybrid language, not purely functional, neither completely object-oriented. Among all the possible functional languages, such as Haskell, Scheme, F#, Clojure, Scala, OCaml/SML, Erlang or Elixir, to name a few; we have chosen Haskell as our functional language.

Some of the reasons for choosing Haskell instead of other more popular functional languages such as Scala, is that Haskell possesses all the properties of a pure functional language and is the go-to language when considering functional programming. These properties are: it is based around functions, which are pure (the same input always produces the same output), first-class variables that can be passed as arguments; and it implements referential transparency, which implies that functions have no side-effects.

Moreover, there are studies which have compared several languages [1], but neither Haskell nor Python have a clear advantage over the other on generic tasks. Thus, this study will try to obtain results over their performance and scalability based on some key metrics in order to choose the better language in the specific field of machine learning.

## 1.2. Document Structure

The document will be structured in four main chapters:

- **Background** - This chapter will cover theoretical information required to comprehend the rest of the study. It will start with the programming languages to be used (Python and Haskell), followed by a brief explanation of machine learning, its taxonomy and then going into more detail regarding the algorithms that are going to be used in this study.
- **Design and Implementation** - In this chapter, we will explain how the test platform will be set up, including the different datasets to be used, how we will pre-process those datasets, and how the chosen algorithms have been implemented. One last section will include some scripts used in the process but that did not entirely fit the mentioned structure.
- **Experimental Results** - This chapter will display the testing framework: the process followed to build it and the code that was implemented in order to utilize the algorithms. The experimental results obtained by applying these algorithms to the datasets will be included so the two programming languages can be compared with evidence of their performance.
- **Conclusion** - The closing chapter will include a conclusion based on the results obtained in the previous chapter highlighting the benefits of each language and their most suited uses, as well as possible future work.

# 2

# Background

In this chapter, we will review background information necessary to follow the rest of the study, starting with an overview of the programming languages that will be compared (Python and Haskell), followed by an explanation of machine learning algorithms, focusing on the ones to be tested. We will end the chapter with a section on other relevant tools that will be used.

## 2.1. Programming Languages

### 2.1.1. Python

Python [2] is an interpreted, object-oriented language with dynamic data-typing. It currently is one of the most popular languages as of IEEE Spectrum's ranking [3]. One of the main reasons for this is its clear syntax and readibility, which offer a very easy learning curve. It is also a very portable language. It can be interpreted in many different operating systems and interacts with most third-party languages and platforms.

It also is the main language for AI and machine learning, along with its less popular stats and Big Data specialist: R. The main reason why it is so widely used is the amount of packages available. From data processing with `numpy` and `pandas` to machine learning algorithms with `scikit` [4], `tensorflow` [5] and `pytorch`, as well as data visualization with `matplotlib`.

All of the functions available in these packages make most of the ML tasks fairly accessible to almost anyone. For example, the code for recognizing images of handwritten numbers (from the MNIST dataset) with a 95+ % accuracy fits in under 30 lines. It is also worth noting that there is a large variety of examples available online, which can make an introduction to ML in Python easier than in other languages.

### 2.1.2. Haskell

Haskell [6] is a purely functional programming language. It is statically typed, has type inference and evaluates commands lazily. The statically typed aspect enforces type safety; type inference refers to the automatic detection of an expression's type; lazy evaluation implies that commands are only evaluated once their result is needed, not before, allowing the compiler to better optimize the code.

As a purely functional language, it has referential transparency as one of its properties, which is that one statement will always evaluate to the same result in any context. Its functions have no side effects and can be used as arguments for other functions. This matches the way machine learning operates: a model that needs to be trained following an algorithm, which minimizes an error function. Since the algorithm usually applies the same operation to multiple sets of data, Haskell's higher order functions can be used to perform this task in an intuitive manner. For example:

- `map func lst` would apply the function `func` to each element of the list `lst`.
- `fold func lst` would reduce the list `lst` to one element by applying the function `func` to every pair of elements (it is assumed that `func` receives two arguments and outputs one).

Side effects may sometimes be necessary. For instance, when trying to load data from a file, you want to store that data. The way side effects are expressed is by means of monadic actions, for example, the I/O monad. We could potentially read from a file as a string and apply all the necessary steps to process it and apply the algorithms, but sometimes it is useful to define a data type with fields, so it can be accessed in a friendlier manner.

Functional programming is not restricted to simply using `map` or `fold` functions. It may be of interest to implement some functional equivalents of object-oriented `getters` and `setters` (in languages such as Java), so that they preserve the functional properties and can be composable and without side effects. In order to do so, Haskell has Lenses.

The code for Lenses is as follows:

**Code 2.1:** Haskell Code for Lenses

```
1  type Lens' a b = forall f . Functor f => (b -> f b) -> (a -> f a)
2
3  view :: Lens' a b -> a -> b
4
5  over :: Lens' a b -> (b -> b) -> a -> a
6
7  set :: Lens' a b -> b -> a -> a
8  set lens b = over lens (\_ -> b)
```

In the first line, the type class `Functor` is used. A functor in Haskell is a type which can be mapped over by using `fmap`. Fmap is a generalization of the map function for lists. Functors must follow two

rules:

- Functors must preserve identity morphisms - `fmap id = id`
- Functors preserve composition of morphisms - `fmap (f . g) == fmap f . fmap g`

Examples of Functors are List, Map or Tree.

The two main functions of a Lens are `view` and `over`. The `view` function receives a data type and returns an attribute of it, while `over` receives a function that modifies the value of an attribute ($b \to b$) and applies it to the data type, returning the result.

The `set` function, is a particular case of `over`, in which the way the attribute is modified does not depend on its previous value. We can see its implementation in lines 7 and 8.

Haskell can take advantage of other existing ecosystems such as *R* with HaskellR and *Spark* with the Sparkle project. Despite this, we will focus on solutions implemented exclusively in Haskell, without the dependancy on other programming language ecosystems.

## 2.2. Machine Learning Algorithms

Data Mining is the process of extracting patterns from large data sets. It involves database systems, statistics and machine learning. This study will focus on machine learning and, more specifically on its algorithms.

Machine Learning is the application of Artificial Intelligence (AI) to provide computer programs with the ability to learn and improve their performance by themselves, without any explicit instructions. The way these algorithms work is by feeding them with already existing data or observations so they can analyze them, come up with patterns and infer results. The goal is to make programs that can accurately predict results in almost any sector, from computer vision to marketing strategies.

Depending on the approach taken during training, Machine Learning can be categorized into three main groups:

- **Supervised** - In supervised learning, the training data contains both the attribute values and the output, also known as supervisory signal. Each example or instance is represented as a row containing all the features or attributes. The collection of rows is represented as a matrix. The mathematical model is built by training with instances trying to minimize the error, or distance from the predicted output to the real one. An algorithm is learning to correctly predict the output if the accuracy of the process rises, or the error decreases, during the training process. Depending on the type of the output, there are three kinds of supervised algorithms:
    - Classification - Classification algorithms have a limited number of output values, called *classes*. The goal of the algorithm is to predict which classes a specific input can belong to. Examples of this could be image classification (determining if a picture corresponds to a cat, dog, and so on) or whether a customer will cancel or renew a subscription.
    - Regression - Regression algorithms have continuous outputs. This is the case when trying to predict

stock prices, a person's height or the temperature at a certain time.

- Similarity learning - This type of learning tries to predict how similar or related two objects are. It is used when wanting to rank or provide suggestions on content, visual or voice verification. These models differ from classification or regression models mostly in the desired output. The algorithm used could be the same, but the expected output is the measure of similarity. For example, an image classifyier could predict that an image corresponds to a panda, or it could provide a ranking of different animals that the one in the picture could be.

- **Unsupervised** - Unsupervised learning, as opposed to supervised, does not contain output values. It is up to the program to try to cluster the data based on common patterns that some of its features may share. Some of its uses are for unsupervised learning in statistical density estimation; or dimensionality reduction (by reducing the number of features based on their relevance) of datasets.
- **Active learning** - Active learning is one where the output labels can be obtained by the algorithm by asking a user for input, for example, or by having the system receive feedback from its environment so that it can estimate how well it is performing. This latter is a type of active learning called Reinforcement learning. It is commonly used for autonomous vehicles, robots interacting with their environment, computer games AI, and so on.

The different types of machine learning are not mutually exclusive. There are mixed models that use several models during different parts of the training process to obtain better results. For example, when trying to classify images, the images can be pre-processed using Spatial Transformer Networks [7] to select a relevant portion of the image, as shown in Figure 2.1. Or when using Support Vector Machines, it is possible to use active learning to obtain more labels [8] and then proceed with the training of the algorithm.

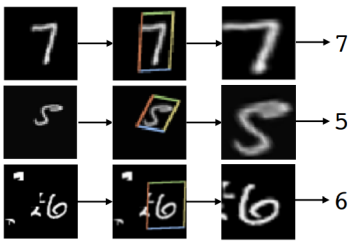



**Figure 2.1:** Spatial Transformer Network MNIST Example

The training process for machine learning algorithms can be suboptimal in different ways. The selected algorithm may not be the best for the specific problem to solve, the available data may not be sufficient or its quality not high enough to allow the algorithm to learn properly. The data related underperformance can fall into two groups: overfitting and underfitting (Figure 2.2).

- **Overfitting** - Overfitting occurs when a model or algorithm adapts to the training set, capturing specific patterns that can be considered as noise. This causes it to not perform as well with new data, as the patterns it is looking for are too specific. It can be reduced through data regularization, which increases the flexibility of the model.
- **Underfitting** - The model cannot properly represent the underlying trend of the data. It can be due to the dataset not being large enough. To reduce underfitting, the regularization parameters should be decreased and, if possible, more features or instances should be added.

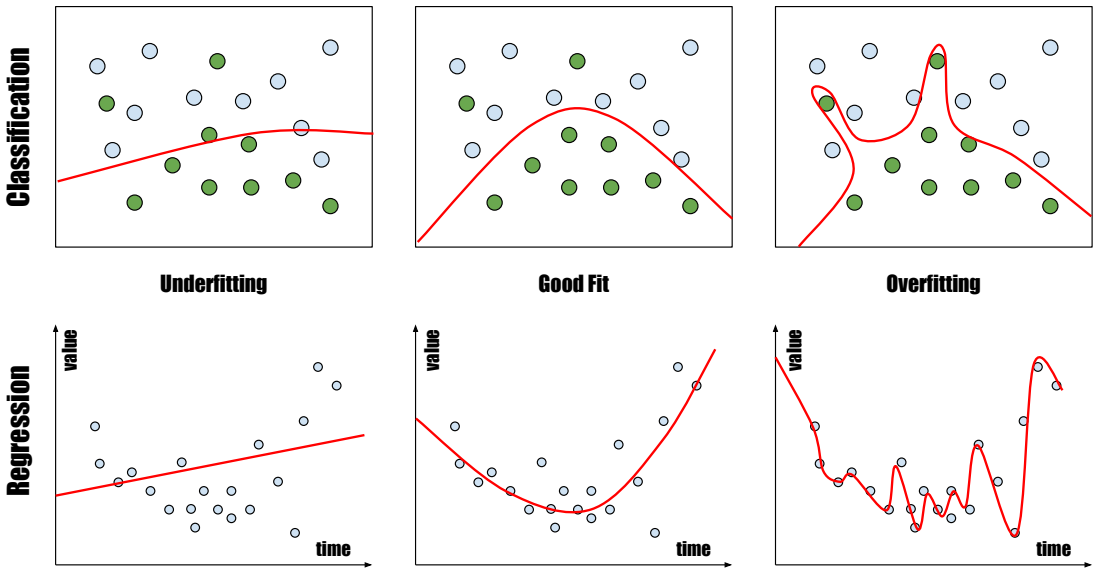



**Figure 2.2:** Overfitting and Underfitting

The machine learning algorithms that we are going to study fall into the category of *Supervised learning*. This study will focus on some of the most used classification algorithms: *Random Forest*, which is based on a *Decision Tree* algorithm; and *Convolutional Neural Networks* (CNNs), which work following the principles of *Neural Networks*.

To that end, we have searched for already existing packages. In Haskell we have found the aima-haskell library [9], which contains the haskell implementation of algorithms as they appear in AIMA [10], including ID3 and the corresponding Random Forest implementation; and also grenade [11], which has an implementation for several Neural Networks.

The Python packages that we will use include scikit-learn and Keras libraries.

### 2.2.1. Supervised Algorithms: Tree Based Classifiers

Decision trees [12] consist of two elements: internal nodes and leaves. Each node corresponds to an attribute or feature and every leaf has a class associated with it.

Each node splits all the elements that fall into it by applying a condition to its feature. For example, in Figure 2.3, depending on the location if it is in the city, suburbs or country, the tree splits the different House instances. The goal of splitting these elements is to end in a leaf node, which contains elements of only one class, in our example, the price of the house. For various reasons, sometimes it is not possible to correctly classify all of the elements, so the leaf's class will be taken as the most common class between all the elements.

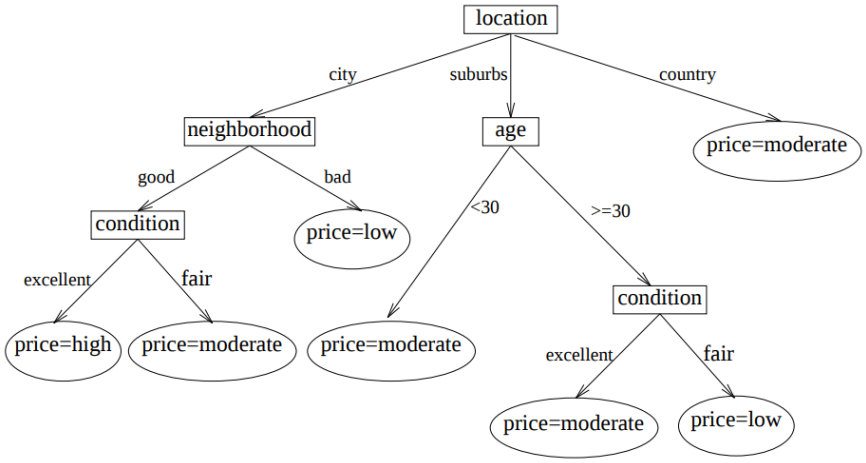



**Figure 2.3:** Decision Tree Example

The aim of decision trees is to provide conditions to split a dataset in a way that minimizes the misclassification rate.

The aima-haskell library provides an implementation of the random forest based on an ID3 decision tree classifier, which we will now explain.

### ID3

The Iterative Dichotomizer 3 (ID3) is an algorithm invented by Ross Quinlan [13] to generate a decision tree.

At each node, between all the attributes of a dataset **S**, it finds the attribute with the highest entropy **H**(S) (or information gain **IG**(S)). It selects the attribute with the lowest entropy (highest information gain) and partitions S into two subsets and then proceeds with the remaining attributes over the newly created nodes.

The algorithm stops when all the elements of a leaf belong to one class or when there are no attributes to split on or elements in a node. The pseudocode is as shown in Algorithm 2.1.

This algorithm is the precursor of the C4.5 [14].

### Random Forest

The random forest algorithm [15] is an ensemble learning method, which means that it uses several algorithms in order to obtain a better predictive performance than the other algorithms alone.

It works by constructing a multitude of Decision Trees that each classify a partition of the data. By using smaller trees, it is less prone to overfitting, which is adapting the algorithm to specifically fit the training set, therefore increasing the error on newer instances. To obtain a final prediction, it calculates

**input :** $Instances, TargetAtt, Attributes$
**output:** $tree$

**if** *all nodes of class* $C$ **then**
  **return** $root$ *with label* $C$;
**if** *Attributes is empty* **then**
  **return** $root$ *with most common label of* $TargetAtt$ *in* $Instances$;
**else**
  att ← attribute with lowest entropy between $Attributes$;
  $root_{decision}$ ← att;
  **foreach** $\mathcal{V}_i$ *of* att **do**
    $instances_{\mathcal{V}_i}$ $leftarrow$ subset of $Instances$ with att value $\mathcal{V}_i$;
    **if** $instances_{\mathcal{V}_i}$ *is empty* **then**
      Add ramification with leaf node with $leaf_{class}$ ← most common value in $TargetAtt$;
    **else**
      Add ramification with subtree `ID3(` $instances_{\mathcal{V}_i}, TargetAtt, Attributes -$ att `)`;

**Algorithm 2.1:** ID3 Algorithm pseudocode

the mode (for classification problems) or the mean (for regression problems) between the values of all the previously trained trees.

### 2.2.2. Supervised Algorithms: Neural Network Based Classifiers

Artificial Neural Networks are computing systems that can *learn* to perform tasks by processing examples. They learn in the sense that they can identify patterns in new data and produce an output accordingly. They receive their name from their biological counterparts: the neural networks located inside animal brains. From this point on, we will refer to Artificial Neural Networks as Neural Networks or NN.

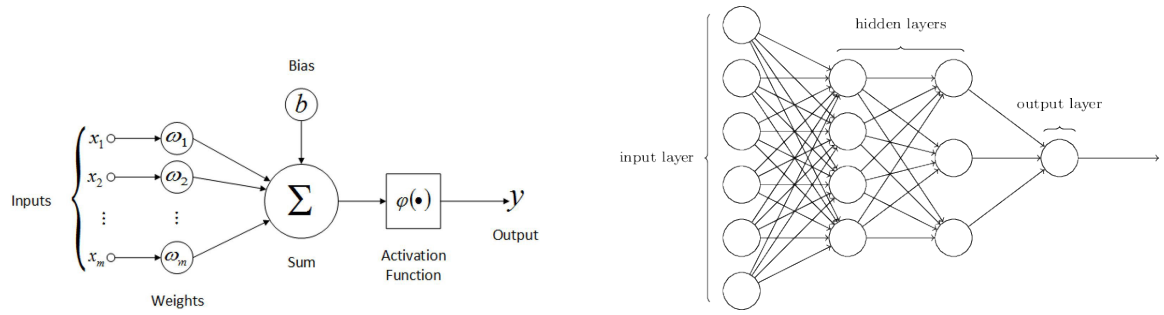
Neural Networks consist of a collection of neurons. Each neuron (Figure 2.4(a)) receives several input values. Each input value is weighted independently, to take into account the difference in importance between all the inputs. This result is then totalled and passed through a non-linear function, such as a sigmoid ($S(x) = \frac{1}{1+e^{-x}}$) or a rectifier ($max(0, x)$), and that is the output of the neuron. The equation that represents how a neuron works is:

$$y = \phi\left(\sum_{i=0}^{m} w_j x_j\right) \tag{2.1}$$

Where $\phi$ is the activation function, $x_i$ with $i \in \{1, 2, ..., m\}$ the inputs and $x_0 = 1$ is considered the bias.

These neurons can be grouped into *layers*. There is an input layer, which takes all the attributes

or features, and an output layer, which provides the end result. Between them, several layers can be added, each performing different kinds of transformations to the input values and thus obtaining more complex patterns. These middle layers are often called *hidden layers* (Figure 2.4(b)).




(a) Artificial Neuron

(b) Network example with two hidden layers

**Figure 2.4:** Neural Network components

The networks are trained by using backpropagation. Backpropagation uses gradient descent to minimize the error of the network by updating the neuron weights (shown in Equation 2.1). If you consider the weights as variables, then you can try to minimize the error $E$, for example the quadratic error $E = \frac{1}{2n}\sum_x \|y(x) - y'(x)\|^2$, where $n$ is the number of training examples, $y(x)$ is the real output and $y'(x)$ is the output the network predicted.

In order to minimize the error, you can move in the direction of maximum change, which is given by its gradient $\nabla E$. The *speed* at which the values move within that direction is given by the *learning rate*: $\eta$. Thus, the final equation for updating the weights is:

$$\Delta w_{ij} = \eta \frac{\partial E}{\partial w_{ij}} \tag{2.2}$$

A more detailed explanation of the process and the algorithm used to compute the derivative appears in [16], Chapter 2.

### Feed Forward Neural Networks

A Feed Forward Neural Network (FFNN), also known as Multilayer Perceptron (MLP), is the simplest type of neural network. It makes use of layers previously mentioned to obtain an output. A key aspect that differentiates it from other types of networks is that the information only flows forward, whereas in networks like Recurrent Neural Networks, loops can be found between layers.

### Convolutional Neural Networks

Convolutional Neural Networks (CNN) are networks often used for image classification due to the spatial properties they possess. Each neuron learns to identify a feature, which is a particular combination of input patterns that will cause it to activate, such as an edge or specific shape.

Instead of fully connected layers, they establish connections through a small window, called *local receptive field*, so that only the neurons within that local receptive field affect the value of a given neuron in the following layer. As we move this window through all the possible positions it can be in, we obtain another layer. This process is shown in Figure 2.5

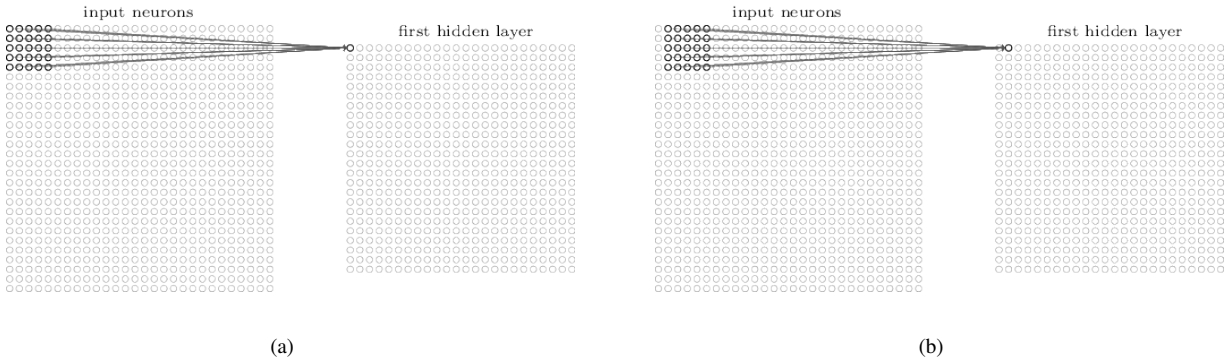



**Figure 2.5:** Convolutional Neural Network's Local Receptive Field

From this point on, we will also refer to CNN's neurons as pixels. The local receptive field can move more than one pixel at a time, and the number of pixels it moves is called *stride* or *stride length*. The weights in this local receptive field are shared, as the goal of the network is to identify features at different locations of the input image (a circle is still a circle, regardless of where in the picture it is placed). The mapping from one layer to another is sometimes referred to as *feature map*. The feature map is defined by a kernel, which consists of the weights and bias. For example, if the feature map size was 5x5, the output of the neuron at the position $(j, k)$ would be:

$$y_{j,k} = \sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m}x_{j+l,k+m}\right)$$

### Pooling Layer

After every convolutional layer, CNNs usually have pooling layers, which simplify the result by condensing the feature map into a smaller one. There are different ways of doing this. For example, it can take regions of 2x2 pixels and output the maximum or mean value of those four pixels. This way, the corresponding feature map of size $n \times m$ will see its size reduced in half: $\frac{n}{2} \times \frac{m}{2}$

To end the CNN, there usually is a FFNN which takes the output of the CNN as input and outputs the predicted class of the instance that is fed to the network.

#### Image Pre-processing

In order to feed images to a CNN, all the images must have the same size in order for the network to be able to process them. There are many factors that can make training CNNs more effective or the end result have a lower error. Some of these factors are:

- **Images of same size** - As mentioned previously, all images be of the same size. To achieve this, images can be expanded or padded on the borders if the size was smaller or reduced in size by resizing or cropping.
- **Reducing size** - Higher quality images have more pixels, which lead to more computations being needed in order to process the same number of images. Thus, reducing the size of the images can be beneficial to reduce training times. These reductions can be:
  - Cropping - Cropping an image does not affect its scale, so it keeps the original proportions. This approach is better if the relevant part of the image corresponds to a smaller portion of it.
  - Resizing - When resizing, you keep most of the information of the image, except its original shape.
- **Threshold** - By applying thresholds to images, certain colors can be made to stand out.
- **Grayscale** - A colored image contains more information. In `RGB` each pixel contains three bytes. Applying a grayscale filter reduces the amount of information to process, thus reducing training times. This may be useful to try if the color is not a determining factor in order to classify the image.

The requirements vary between problems, so images will usually need to be pre-processed in different ways depending on the problem at hand.

## 2.3. Additional Software Tools and Frameworks

In order to perform this study, we will use the following tools:

#### Remote Machines

The tests will be performed in remote machines that will be accessed using the SSH protocol. The goal of this is to produce results that are as accurate as possible and that can be reproduced if required. The accuracy factor is such because when running code on a local machine, the case usually is that there is a fair number of processes running in the background, which may vary the training times for our algorithms. Thus, by running the code on a remote machine, we remove many of these additional processes that use the CPU.

The SSH protocol used to connect with the remote machines is a cryptographic network protocol that allows you to securely connect to a remote SSH-server.

#### Scripts

Scripts will be run in the remote machine using both Bash and Python. Scripts are a collection of commands to be run sequentially. They are used to automate specific tasks by grouping several

commands under the same file, which can then be run with a single command.

#### Version Control

As version control system (VCS), we will use `git`. VCS are tools that allow the management of changes to documents. In our case, it will be used for backing up our project and allowing to have the code updated in several machines with ease, being all the changes accesible for all the devices that have cloned the repository [1]. It has the additional benefit that it stores different versions of the files so changes can be reverted should any of them not be wanted later on.

#### Jupyter Notebook

Jupyter Notebook is an open-source web application that allows the creation of notebooks that include both documentation and runnable code. A primary reason for using it is due to its interactive nature. It is similar to programming on the terminal in the sense that you can easily test new commands, so coding becomes faster. It also lets you comment the code with Markdown cells, making it both easier to read and more appealing.

[1] Cloning a repository is making a copy of a project in a device.

# 3 Design and Implementation of a Test Bench Software Platform

The desired testing platform has an overview which is shown in Figure 3.1. This layout consists of different modules.

- **Datasets** - The data which contains the information of the instances to classify.
- **Pre-processing** - Module that fetches the information from the datasets and transforms it into a format which is compatible or better suited for the algorithm.
    - For datasets that will be processed by tree classifying algorithms, numerical the data needs to be split into different categories.
    - For FFNN, all the inputs must be numerical, so all categories and other non-numerical features must be converted.
    - In CNN, the pre-processing is particularly important, as with images there are many options for modifying the values of the images. For example, applying filters to highlight features, resizing or adding more images using Data Augmentation. [17].
- **Train-Test Split** - Splitting the datasets into a training and a testing subset can be done by the program running the algorithm or as a previous step.
- **Algorithm** - This module contains the code to run the algorithm, including both the algorithm code and lines to feed it the corresponding dataset.
- **Test Bench** - Module that is in charge of running the algorithms with specific parameters and record the time they take and their CPU and memory usage, so the data can be extracted later on to produce relevant results, both numeric (tables) and visual (graphs).

In this chapter we will try to follow these modules, starting with datasets and then continuing to pre-processing, algorithm implementation and concluding some scripts that will help create new Haskell files for tree classification and measure the performance.

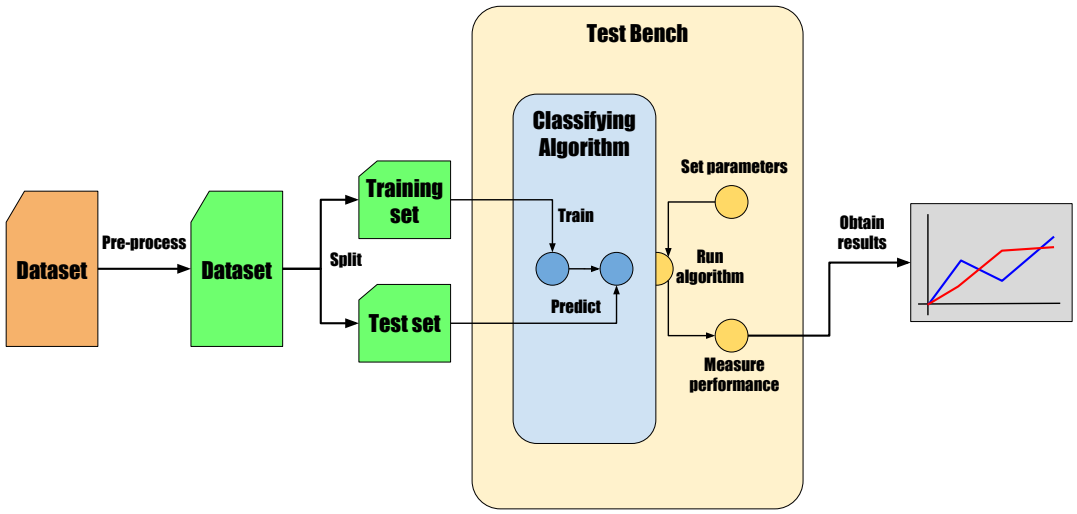



**Figure 3.1:** Testing Platform Overview

## 3.1. Datasets

We have used different kinds of datasets depending on the algorithm to test. For example, datasets for ID3 trees should be categorical (which means that every attribute has a fixed number of unique values); FFNNs should receive numerical values, although there are restrictions for the haskell code, as the output cannot be continuous; and CNNs should receive images as input.

Given the difficulty to find datasets that are purely as mentioned above, we will need to apply some pre-processing (Section 3.2) to get all the values to be consistent with the algorithms to be used.

### 3.1.1. Datasets for Tree Classifiers

For tree classifiers we wil use the *1985 Automobile Dataset* [1], *Balance Scale* [2], *Student Performance* [3] and *Bank Marketing* [4] datasets.

[1] 1985 Automobile Dataset - https://www.kaggle.com/fazilbtopal/auto85

[2] Balance Scale Dataset - https://www.kaggle.com/mysticvalley/balance-scale

[3] Student Performance Dataset - https://www.kaggle.com/spscientist/students-performance-in-exams

[4] Bank Marketing Dataset - https://www.kaggle.com/henriqueyamahata/bank-marketing

| Dataset | Description | Number of Attributes | Number of Instances |
|---|---|---|---|
| 1985 Automobile Dataset | Dataset with information about cars and their associated risk rating. | 15 continuous<br>1 integer<br>10 categorical } = 26 | 205 |
| Balance Scale | Dataset with columns for right and left weight and distance. The class to predict is whether the scale is balanced, tilted to the right or to the left. | $4 \in \{1, 2, 3, 4, 5\}$ | 625 |
| Student Performace | Marks secured by the students based on some values such as race/ethnicity or parental level of education. The grades are divided into *math*, *reading* and *writing* scores. | 5 categorical | 1000 |
| Bank Marketing | The data is related with direct marketing campaigns (phone calls) of a Portuguese banking institution. The classification goal is to predict if the client will subscribe a term deposit (variable y). | 10 numerical<br>10 categorical } = 20 | 41188 |

**Table 3.1:** Datasets to be used for Tree classifying algorithms

### 3.1.2. Datasets for FFNN Classifiers

The datasets used for FFNN are: *Graduate Admissions* [5] and *Mice Protein Expression* [6].

| Dataset | Description | Number of Attributes | Number of Instances |
|---|---|---|---|
| Graduate Admissions | Dataset to predict the chance of admitting students for their Master's Degree depending on their scores, university and so on. | 9 numeric | 500 |
| Mice Protein Expression | Dataset with measurements of different proteins to assess the effect of the drug memantine in recovering the ability to learn in trisomic mice. | 77 continuous<br>3 categorical } = 80 | 1080 |

**Table 3.2:** Datasets to be used for FFNN classifiers

### 3.1.3. Datasets for CNN Classifiers

For the Convolutional Neural Network, we will use the *Malaria Cell Images Dataset* [7], with differences in the number of images processed.

[5] Graduate Admissions - https://www.kaggle.com/mohansacharya/graduate-admissions

[6] Mice Protein Expression - https://www.kaggle.com/ruslankl/mice-protein-expression

[7] Malaria Cell Images Dataset - https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria

| Dataset | Description | Number of Images | Image Size |
|---|---|---|---|
| Malaria Cell Images Dataset | Dataset to predict whether a cell contains Malaria or not. | 13779 Uninfected<br>13779 Infected<br>} = 27,558 total images | Varied sizes<br>70-232x82-235 |

**Table 3.3:** Datasets to be used for CNN classifiers

## 3.2. Pre-processing

The algorithms, in addition to the existing implementation of them, impose some constraints in the way data is fed to them. In this section we will analyze the specific pre-processing that has been followed for each algorithm.

### 3.2.1. Dataset Pre-processing for Tree Classifiers

The tree classifier that we will use is the ID3. This type of tree receives only attributes with categorical values. To do so, we have used a Python script which includes the lines in Code 3.1.

**Code 3.1:** Creating bins to categorize numerical data

```
nbins = 10

for att in df.columns:
    if (df[att].dtype == np.float64 or (df[att].dtype == np.int64 and len(df[att].unique()) > 10)):
        df[att] = pd.cut(df[att], bins=nbins, labels=range(nbins))
```

The script is intended to create ten bins in which the data is fitted. For example, if there were values from 1 to 100, the script would return that the first ten belong to the bin 1, the next 10 to the bin 2, and so on.

As the Haskell ID3 program requires the use of attribute names, we will also rename the attributes that are Haskell keywords. A few of these examples could be: *length* to *attLength* or *class* to *target*.

Another issue that has to be addressed is that the categorical values in Haskell are defined as *enumerations*, which means every attribute should be capitalized and the values for them cannot be reused. For example, if there were two attributes:

$$height \in \{low, medium, high\} \text{ and } size \in \{Xsmall, small, medium, large, Xlarge\}$$

the value for $medium$ would not be properly defined. Therefore, we also modify the values prepending the attribute name. An example of dataset pre-processing is shown in Table 3.4.

| buying | maint | doors | persons | lug_boot | safety | class |
|---|---|---|---|---|---|---|
| vhigh | vhigh | 2 | 2 | small | low | unacc |
| high | vhigh | 5more | more | big | high | unacc |
| high | high | 2 | 2 | small | low | unacc |
| high | high | 2 | 4 | med | high | acc |
| med | vhigh | 5more | 4 | med | low | unacc |

(a) Original Dataset.

| buying | maint | doors | persons | lug_boot | safety | target |
|---|---|---|---|---|---|---|
| BuyingVhigh | MaintVhigh | Doors2 | Persons2 | LugBootSmall | SafetyLow | TargetUnacc |
| BuyingHigh | MaintVhigh | Doors5more | PersonsMore | LugBootBig | SafetyHigh | TargetUnacc |
| BuyingHigh | MaintHigh | Doors2 | Persons2 | LugBootSmall | SafetyLow | TargetUnacc |
| BuyingHigh | MaintHigh | Doors2 | Persons4 | LugBootMed | SafetyHigh | TargetAcc |
| BuyingMed | MaintVhigh | Doors5more | Persons4 | LugBootMed | SafetyLow | TargetUnacc |

(b) Modified Dataset.

**Table 3.4:** Example of pre-processing a dataset requires to work on aima-haskell's ID3 algorithm

### 3.2.2. Dataset Pre-processing for FFNN Classifiers

The grenade Haskell code for Neural Networks expects two input files: one for training and one for testing. If there is a reason for the test set to be fixed, this pre-processing split presents no major inconvenience. Nevertheless, it could present some problems. When aiming to reduce a model's error, the specific way the dataset is shuffled could impact the results. Hence, it is convenient to test the model with different random training and testing set to obtain a more accurate estimation of the error. This would imply saving different training and testing files every time the model is run, reducing the efficiency.

In our study, the error of the models is not relevant, in the sense that we want to measure how efficient they are in processing a certain dataset, so we can save the files once and reuse them as needed.

The output class is required to be the first column of the dataset; therefore, we will reorder the columns should they be in a different order.

### 3.2.3. Dataset Pre-processing for CNN Classifiers

For the pre-processing of images, we will take the pre-processing considerations expressed in Subsection 2.2.2. We have used the lines in Code 3.2 to show one possible pre-processing flow.

**Code 3.2:** Code to pre-process images

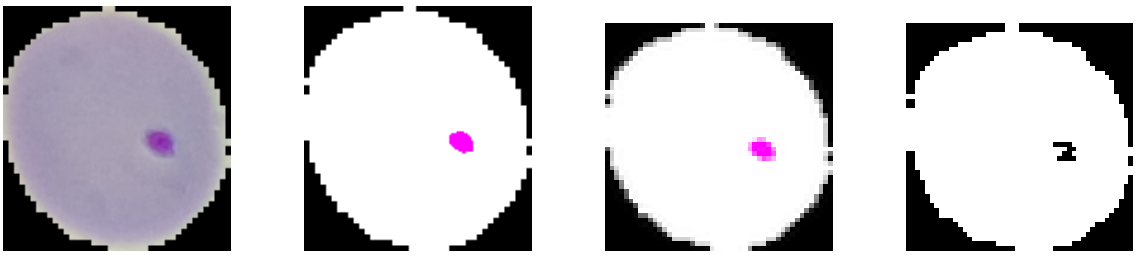
```
im = Image.open(file)

# Apply the threshold to create a higher contrast
threshold = 100
im = im.point(lambda p: p > threshold and 255)

# Convert the image to grayscale
im = im.convert('1')

# Resizing the image
size = 48,48
im = im.resize(size, Image.ANTIALIAS)
```

Depending on whether we wish to train the network with grayscaled images or colored; or with different sizes to increase the resolution, we may slightly vary or omit these pre-processing layers. Figure 3.2 displays how the previous code modifies the original image.



(a) Normal Malaria Cell (b) Cell after applying threshold (c) Cell after resizing (d) Cell after grayscaling

**Figure 3.2:** Malaria Cell Processing

After having pre-processed the images, we proceed to flattening them to write them down to the training and testing files on a csv format that is readable by the scripts that will run the algorithms.

## 3.3. Implementation

In this section we will give details about how the code is implemented, starting with tree based and then proceeding to neural based classifying algorithms. We will include the additional code we have required to be able to read the datasets from files and feed them to the algorithms to receive an output from them.

### 3.3.1. Tree based algorithms: Haskell ID3 implementation

For tree based algorithms, we have used the `aima-haskell` [9] library obtained from GitHub for the Haskell algorithm

There are three main aspects of the Haskell implementation: Data Reading and Handling, Data Types and Attribute Declaration, and the main code.

#### Data Reading and Handling

We have decided to use the `Frames` Haskell package, which is an efficient tool for working with tabular data files. Its performance is comparable to pandas, the popular Python library for DataFrames [18]. Frames work by having records as rows. These records are implemented in Vinyl, and they allow rows to have different types. Its implementation is based on the data type `Rec` for records. It achieves a high performance by making them array-backed (`ARec`), so that the access time can be reduced from $\mathcal{O}(n)$ to $\mathcal{O}(1)$. The way attributes are accessed is using Lenses, as explained in Section 2.1.2.

**Code 3.3:** Haskell ID3 Data Manipulation for the Balance Scale Dataset

```
1  tableTypes "BalanceScale" "../data/balance-scale/balanceScale_mod.csv"
2
3  data BalanceScaleTarget = TargetB | TargetL | TargetR deriving
       (Show,Eq,Ord,Enum,Bounded,Read)
4  data BalanceScaleLWeight = LWeight1 | LWeight2 | LWeight3 | LWeight4 | LWeight5 deriving
       (Show,Eq,Ord,Enum,Bounded,Read)
5  data BalanceScaleLDistance = LDistance1 | LDistance2 | LDistance3 | LDistance4 | LDistance5 deriving
       (Show,Eq,Ord,Enum,Bounded,Read)
6  data BalanceScaleRWeight = RWeight1 | RWeight2 | RWeight3 | RWeight4 | RWeight5 deriving
       (Show,Eq,Ord,Enum,Bounded,Read)
7  data BalanceScaleRDistance = RDistance1 | RDistance2 | RDistance3 | RDistance4 | RDistance5 deriving
       (Show,Eq,Ord,Enum,Bounded,Read)
8
9
10 atts :: [Att BalanceScale]
11 atts = [ att (\x -> read T.unpack rget lWeight x :: BalanceScaleLWeight) "LWeight"
12          , att (\x -> read T.unpack rget lDistance x :: BalanceScaleLDistance) "LDistance"
13          , att (\x -> read T.unpack rget rWeight x :: BalanceScaleRWeight) "RWeight"
14          , att (\x -> read T.unpack rget rDistance x :: BalanceScaleRDistance) "RDistance"]
15
16 balanceScaleTargetAtt :: Att BalanceScale
17 balanceScaleTargetAtt = att (\x -> read T.unpack rget target x :: BalanceScaleTarget)
       "TargetBalanceScale"
18
19 balanceScaleStream :: Producer BalanceScale IO ()
20 balanceScaleStream = readTableOpt defaultParser "../data/balance-scale/balanceScale_mod.csv"
21
22 loadBalanceScale :: IO (Frame BalanceScale)
23 loadBalanceScale = inCoreAoS balanceScaleStream
```

The Frame is declared as a data type in the first line of the Code 3.3. The way it is used is:

```
tableTypes "FrameName" "path/to/source/file"
```

This line creates the data type and lenses required to access its attributes, which will be the column names. For filling the dataset with values, there are two functions: one to create a stream and the other to fill the Frame from the stream. In the example (Code 3.3), these lines appear from line 19 to 23.

### Data Types and Attribute Declaration

Data types are declared following the structure:

```
data AttributeName = AttValue1 | AttValue2 | ... | AttValueN
          deriving (Show,Eq,Ord,Enum,Bounded,Read)
```

The deriving part is required so that Haskell automatically implements the necessary functions to compare and display the values of the enumeration of attributes.

An example of data type declaration occurs in Code 3.3, from lines 3 to 7.

In order to create attributes, we use a lambda function [8] that gets a row of the frame and calls the corresponding Frame function to obtain that specific attribute. Since attributes are parsed as text, it is required to parse the data into its corresponding data type.

An example of declaration of attributes appears in lines 10-17. The target attribute is treated separately because it will not be processed by the ID3 algorithm in the same way.

### Main

For the main code, the steps are: loading the dataset, splitting it into training and test sets with the functions we created in module FrUtil, creating a tree with the training set and obtaining the error with the test set. It is exemplified with the Balance Scale dataset as shown in 3.4.

**Code 3.4:** Haskell ID3 Main for the Balance Scale Dataset

```
1  dataset <- loadBalanceScale
2  let (trainSet,testSet) = trainTestSplit dataset 0.7 seed
3  let tgt = test balanceScaleTargetAtt
4
5  let tree = fitTree tgt atts trainSet
6
7  let error = mcr (DT.decide tree) testSet (fmap tgt testSet)
8
9  putStrLn ("The error rate was: " ++ show error)
```

At an early point of the study, we also considered adapting the ID3 and Random Forest code to use

[8] Anonynous function inside the code

Frames instead of lists. We proceeded to write all the code necessary for this adaptation, but the results we obtained were almost twice the time the list version took, so from that point on, we used that version instead.

### 3.3.2. Tree based algorithms: Haskell RF implementation

For the Random Forest implementation, there are not many changes to be made to Code 3.4. Instead of creating a tree, we create a forest with similar parameters:

**Tree** `let tree = fitTree tgt atts trainSet`

**Forest** `let forest = unsafePerformIO $ evalRandIO $ randomForest ntrees natts tgt atts trainSet`

The second change is to modify the decision function to predict from is the tree `decide` to the forest `decide`, so the final result looks like this:

**Tree** `let error = mcr (DT.decide tree) testSet (fmap tgt testSet)`

**Forest** `let error = mcr (RF.decide forest) testSet (fmap tgt testSet)`

### 3.3.3. Tree based algorithms: Python ID3 implementation

In order to use the same algorithm in Python, we will use the `decision-tree-id3` implementation of the ID3 algorithm, instead of the `scikit-learn` implementation of the decision tree, which uses an optimized version of the CART algorithm (Classification and Regression Models) instead of the ID3.

Its implementation is shown in the code extract 3.5.

**Code 3.5:** Python ID3 Classifier

```
1 estimator = Id3Estimator()
2 estimator.fit(X_train, y_train)
3 y_pred = estimator.predict(X_test)
4 print("The model's accuracy is: " + str(accuracy_score(y_test, y_pred)))
```

### 3.3.4. Tree based algorithms: Python RF implementation

For the Random Forest, as we wish to use the previous implementation of ID3, we have used an already existing random forest implementation [9], and adapted it to use `decision-tree-id3`'s ID3.

[9] Random forests and decision trees from scratch in Python

The code appears in Code 3.6.

**Code 3.6:** Python RF Classifier

```
class RandomForest():
    def __init__(self, x, y, n_trees, n_features, sample_sz = -1):
        np.random.seed(12)

        self.x, self.y, self.sample_sz = x, y, sample_sz
        self.trees = [self.create_tree() for i in range(n_trees)]

    def create_tree(self):
        return Id3Estimator()

    def fit(self):
        for tree in self.trees:
            idxs = np.random.permutation(len(self.y))[:self.sample_sz]
            f_idxs = np.random.permutation(self.x.shape[1])[:self.n_features]
            tree.fit(self.x.iloc[idxs], self.y.iloc[idxs])

        return self

    def predict(self, x):
        matrix = np.array([t.predict(x) for t in self.trees])
        most_common = lambda l: np.argmax(np.bincount(l))
        return np.apply_along_axis(most_common, 0, matrix)

df = pd.read_csv(filename)

for col in df.columns:
    df[col] = df[col].astype("category").cat.codes

X = df[df.columns[1:]]
y = df[df.columns[0]]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

forest = RandomForest(X_train, y_train, ntrees)
forest.fit()
yf_pred = forest.predict(X_test)
print("The model's accuracy is: " + str(accuracy_score(y_test, yf_pred)))
```

The implementation primarily creates many ID3 trees, fits them with random partitions of data and predicts using the mode between them all. The main code is similar to the one for the single ID3 tree.

### 3.3.5. NN based algorithms: Haskell FFNN implementation

For Neural Networks in Haskell, we have used the `grenade` [11] library for the simplicity it offers when creating different networks.

The only two data types that have to be defined for each dataset are the input and output sizes

in `NetShape`, and in `Network`, the different layers and the sizes. For the Mice Protein Expression dataset, an example network with a hidden layer of 40 neurons appears in Code 3.7

**Code 3.7:** Mice Protein Expression Haskell Network Example

```
type NetShape = (S ('D1 80), S ('D1 8))

type NET
  = Network
    '[ FullyConnected 80 40, Logit, FullyConnected 40 8, Logit]
    '[ 'D1 80, 'D1 40, 'D1 40, 'D1 8, 'D1 8]
```

### 3.3.6. NN based algorithms: Haskell CNN implementation

CNNs have a very similar implementation, but the layers we will use are the ones proper to a CNN, as explained in Section 2.2.2. An example for the Malaria Cell Images Dataset network layout, for colored images of size 32x32, appears in 3.8.

**Code 3.8:** Malaria CNN Example

```
type NetShape = (S ('D3 32 32 3), S ('D1 2))

type NET
  = Network
    '[ Convolution 3 10 5 5 1 1, Pooling 2 2 2 2, Relu
     , Convolution 10 16 5 5 1 1, Pooling 2 2 2 2, Reshape, Relu
     , FullyConnected 400 80, Logit, FullyConnected 80 2, Logit]
    '[ 'D3 32 32 3, 'D3 28 28 10, 'D3 14 14 10, 'D3 14 14 10
     , 'D3 10 10 16, 'D3 5 5 16, 'D1 400, 'D1 400
     , 'D1 80, 'D1 80, 'D1 2, 'D1 2]
```

The network specified on top consists of two convolutional layers, each followed by a pooling layer and in the end two fully connected layers that lead to the output layer, which is formed by two neurons.

### 3.3.7. NN based algorithms: Python FFNN implementation

For the FFNN in Python, we will use the `scikit-learn` package.

The example in Code 3.9 is the Python equivalent to the Haskell code that appears in Code 3.7.

**Code 3.9:** Mice Protein Expression Python Network Example

```
net = MLPClassifier(hidden_layer_sizes=(40), activation='logistic', max_iter=iters)
```

In comparison, the Python code is more compact. It is also easier to add more hidden layers because it is done by adding one additional number.

### 3.3.8. NN based algorithms: Python CNN implementation

The CNN python package that we will use is `Keras`, which is based on the `TensorFlow` framework. The example Code 3.3.8 adds the same layers to the network as were added in Code 3.8.

**Code 3.10:** Malaria Python CNN Example

```
model = Sequential()

model.add(Conv2D(10, kernel_size=5, padding="same",input_shape=inshape, activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(16, kernel_size=5, padding="same", activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Flatten())
model.add(Dense(units=80, activation='relu'))
model.add(Dense(2))
model.add(Activation("softmax"))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

The Python code for designing a particular CNN may span more lines, but it is not required to write the sizes for all of the layers, which reduces the number of necessary changes when adjusting hyperparameters. Hyperparameters are the parameters that are set before the learning process begins and are not derived from the training.

By contrast, if there is an error with the network, the Haskell code will notice the error when compiling the code, instead of producing a Runtime Error like Python does.

## 3.4. Scripts

We have created several scripts to automate some processes which were overly repetitive. For example, the creation of data types for decision trees and random forests in Haskell. As they have to be hardcoded, we implemented a script that, from a *csv* file, would output an *hs* file that could be run. The way this process is implemented is via Jupyter Notebook, and its name is: *gen_hs_file.ipynb*, which can be found in the GitHub repository of the project.

The script reads the *csv* file and uses the attribute names and values as data type name and enumeration values, respectively. It also modifies the *csv* file to that it matches what the *hs* file will expect and creates the Frame structure so the different fields can be accessed. All values read from the *csv* file are cleaned from special symbols that can affect the proper functioning of the Haskell program, for example '.', which is the composition symbol in Haskell, or '-' and '/' which are considered operators.

The other script implemented is a performance script displayed below. With it, we will record the CPU and Memory usage of the programs we launch:

**Code 3.11:** Bash Performance Script

```
#!/bin/bash

DIR=$(dirname "$(readlink -f "$0")")
output=$1
shift
cd $(pwd) && $* &
pid=$(pgrep -f $1 | tail -n 1)
cd $DIR
while ps -p $pid > /dev/null
do
        ps -p $pid -o pcpu= -o pmem= >> $output
        sleep 1
done
```

# 4

# Experimental Results

In this chapter we will present the different tests performed, along with their results. We will start with the remote machine where all the tests were run. After that, we will proceed with the tests for tree classifying algorithms varying the number of trees from one (decision tree) to 1000. Then we will display the results of varying the number of iterations for FFNNs for the two datasets, with two different network layouts for the Mice Protein Expression Dataset. We will conclude the chapter with results of an incremental set of the Malaria Cell Dataset (100, 1k and 10k images) with varying number of iterations.

## 4.1. Machine Specifications

The remote machine on which we will run the tests has the following specifications:

- Operating system - CentOS Linux 7 (Core)
- Kernel version - Linux 3.10.0-693.21.1.el7.x86_64
- CPU model - Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
- Number of processors - 48
- Memory - 126 GB

## 4.2. Experimental Results: Tree Classifiers

The Table 4.1 has the time and memory usage of the corresponding Haskell and Python programs. The memory value is the memory percentage (rounded to the first decimal, so any value under 0.05 % turns into 0.0) that the Operating System allocates to that specific process.

For a graphic comparison of both programming languages, we can observe Figure 4.1. As the results show, the execution of the Haskell implementation is significantly slower than that in Python. With the smaller datasets, it goes from around 3-5 times slower with few trees to being 20 times slower in the Student Performance Dataset with 1000 trees. When looking at the Bank Dataset, we see the difference going up to 45 times slower with just 10 trees.

| Dataset | # trees | Haskell | | Python | |
|---|---|---|---|---|---|
| | | Time (s) | Memory ( %) | Time (s) | Memory ( %) |
| Auto 85 | 1 | 8 | 0.3 | 2 | 0.0 |
| | 10 | 17 | 0.3 | 2 | 0.0 |
| | 100 | 123 | 0.3 | 14 | 0.0 |
| | 1000 | 1174 | 0.3 | 128 | 0.1 |
| Balance Scale | 1 | 2 | 0.1 | 1 | 0.0 |
| | 10 | 3 | 0.1 | 1 | 0.0 |
| | 100 | 13 | 0.1 | 3 | 0.0 |
| | 1000 | 113 | 0.2 | 14 | 0.1 |
| Student Performance | 1 | 2 | 0.1 | 1 | 0.0 |
| | 10 | 7 | 0.1 | 2 | 0.0 |
| | 100 | 57 | 0.1 | 4 | 0.0 |
| | 1000 | 556 | 0.4 | 26 | 0.1 |
| Bank | 1 | 176 | 0.4 | 6 | 0.0 |
| | 10 | 1777 | 0.6 | 39 | 0.1 |
| | 100 | 18869 | 2.0 | 360 | 0.9 |

**Table 4.1:** Tree Classifying Algorithms Performance Comparison

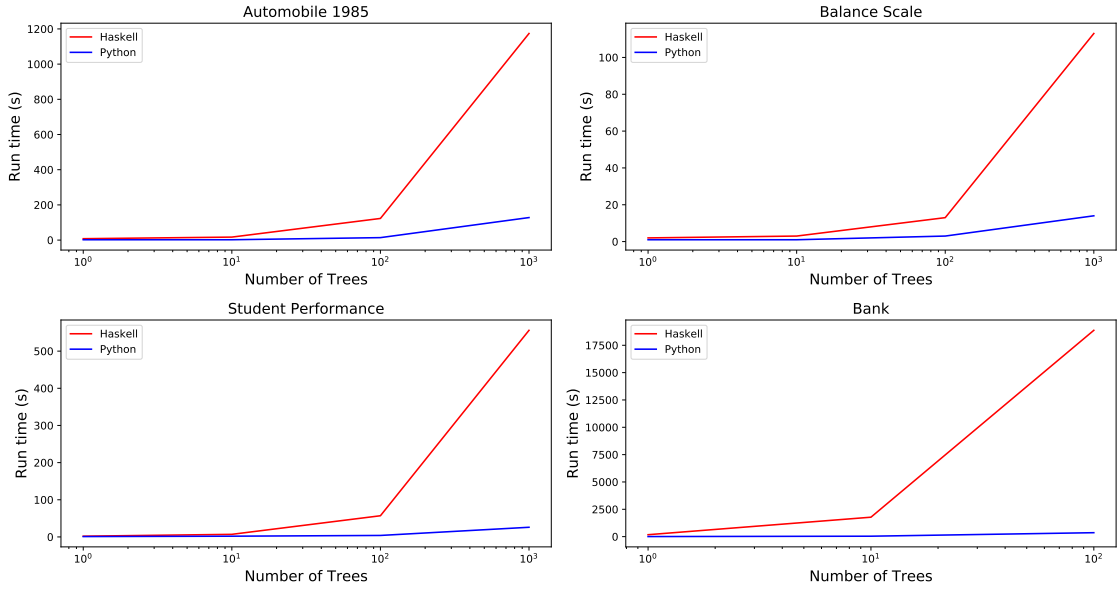



**Figure 4.1:** Tree Performance Comparison Graphs

## 4.3. Experimental Results: FFNN Classifiers

When running the algorithms on the datasets, we encountered several setbacks. Firstly, the Graduate Admissions Dataset is not viably classified with this algorithm. Both networks have a class prediction accuracy of under a third and they tend to stop improving after a certain point. The `MLPClassifier`, by default, stops training once the algorithm does not improve its score in two consecutive operations. This is intended to increase the efficiency of the algorithm by stopping once training stops being useful. Nevertheless, since its Haskell counterpart does not have such functionality, we had to change the *tolerance for optimization* parameter: `tol`. This is what we expect to achieve by setting the parameter to 0.

The second setback was realizing that the `MLPClassifier` uses more than one core by default. We observed this when the CPU usage for the Python scripts was around 600 % in some cases. On the other hand, the Haskell FFNN only uses a single core, so the comparison would not be fair. To fix that, we have modified the performance script by adding the taskset command. Its usage is as follows:

```
taskset -c n1,n2,... command
```

This way, the `command` will be executed in the cores with numbers n1, n2, and so on. If we only specify one core (for example: `taskset -c 3 python...`), the process is assigned to that core exclusively.

After applying these changes, we have run the algorithms and obtained the experimental results which are shown in the Table 4.2.

| **Dataset** | **# trees** | **Haskell** | | **Python** | |
|---|---|---|---|---|---|
| | | Time (s) | Memory ( %) | Time (s) | Memory ( %) |
| Graduate Admissions | 10 | 8 | 0.1 | 2 | 0.0 |
| | 100 | 17 | 0.1 | 2 | 0.0 |
| | 1000 | 123 | 0.1 | 14 | 0.0 |
| | 10000 | 1174 | 0.4 | 128 | 0.1 |
| Mice Protein Expression | 10 | 2 | 0.2 | 1 | 0.0 |
| | 100 | 4 | 0.2 | 2 | 0.0 |
| | 1000 | 26 | 0.2 | 10 | 0.0 |
| | 10000 | 502 | 0.2 | 37 | 0.0 |

**Table 4.2:** FFNN Performance Comparison

Seeing that the Graduate Admissions Dataset is not appropriate, we have tested the Mice Protein Expression Dataset once more, changing the Neural Network layout from one 40-neuron hidden layer to two (80,40)-neuron hidden layers. The comparison appears in Table 4.3.

The results point to a similar pattern as with the tree algorithms. As the number of iterations grows, Haskell's training times do so at a much higher rate than Python's, needing over 10 times as much time to finish running.

| Dataset | # Iterations | Haskell | | Python | |
|---|---|---|---|---|---|
| | | Time (s) | Memory (%) | Time (s) | Memory (%) |
| Mice Protein Expression 2 Hidden Layers | 10 | 2 | 0.2 | 3 | 0.0 |
| | 100 | 7 | 0.2 | 4 | 0.0 |
| | 1000 | 61 | 0.2 | 24 | 0.0 |
| | 10000 | 1337 | 0.2 | 111 | 0.0 |

**Table 4.3:** 2 hidden layers FFNN Mice Performance Comparison

The graphs for the previous tests are shown in Figure 4.2

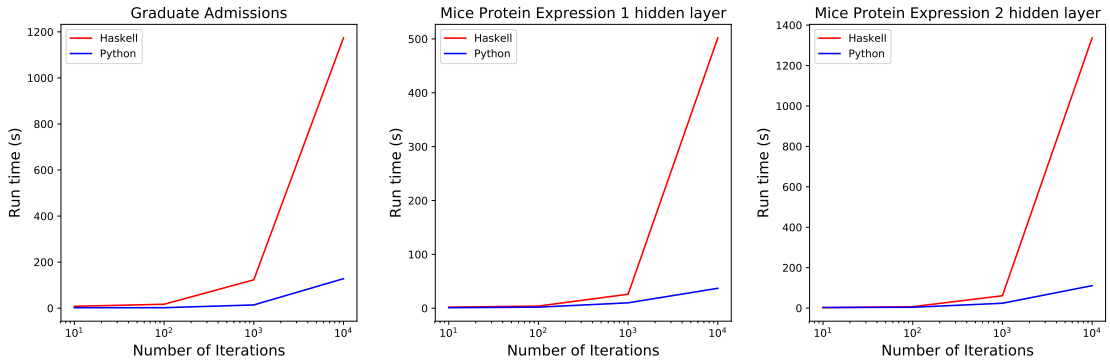



**Figure 4.2:** FFNN Performance Comparison Graphs

## 4.4. Experimental Results: CNN Classifier

For the CNN, we have used a larger dataset, compared to the others. In order to see how well the algorithms scale, we have created three different partitions of this dataset. One with 100 images, another with 1k and the last one with 10k images. We have tested the performance of the network described in Code 3.8 and 3.10 (for Haskell and Python, respectively) for a different number of epochs [1], or iterations: 10, 100 and 1000. The results appear in Table 4.4.

[1] An epoch is one forward and backward pass of all training examples.

| Number of Images | # Iterations | Haskell | | Python | |
|---|---|---|---|---|---|
| | | Time (s) | Memory ( %) | Time (s) | Memory ( %) |
| 100 | 10 | 3 | 0.3 | 5 | 0.1 |
| | 100 | 20 | 0.3 | 15 | 0.1 |
| | 1000 | 178 | 0.3 | 128 | 0.1 |
| 1000 | 10 | 20 | 0.3 | 16 | 0.1 |
| | 100 | 182 | 0.3 | 119 | 0.1 |
| | 1000 | 1919 | 0.3 | 1146 | 0.1 |
| 10000 | 10 | 190 | 1.2 | 131 | 0.5 |
| | 100 | 1687 | 1.2 | 1203 | 0.5 |
| | 1000 | 16414 | 1.2 | 11632 | 0.5 |

**Table 4.4:** CNN Malaria Cell Performance Comparison

For the CNN, as we can see, the performance difference is not quite as large as with other algorithms. Haskell performs around 1.5 times slower than Python, although the memory reserved is 2-3 times as much.

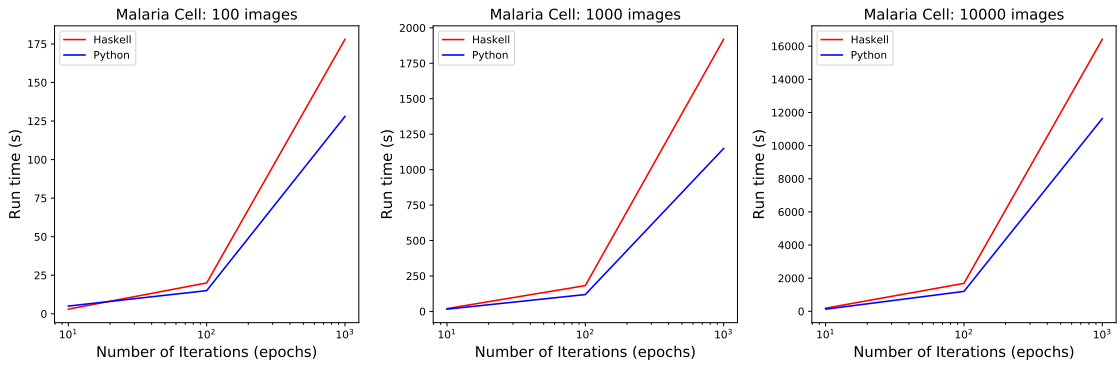



**Figure 4.3:** CNN Performance Comparison Graphs

# 5

# Conclusion

This project has focused on determining whether Haskell is a good alternative to Python in the field of Data Mining and Machine Learning. To achieve this goal, we implemented a test bench software platform that tested two of the most commonly used algorithms: Random Forest and Convolutional Neural Networks; both in Python and Haskell, using the libraries available for each language: `decision-tree-id3`, `scikit-learn` and `TensorFlow` for Python and `aima-haskell` and `grenade` for Haskell.

The goal of this study was not to compare the accuracy obtained in each programming language, as that should depend mostly on the algorithm used, but to obtain values about how scalable each language is in terms of the resources it needs and the time it takes for a given algorithm to process a dataset. The results were widely favorable to Python for both the trees and the FFNN, having a performance which is around ten times faster than Haskell, even more as the datasets became larger. Despite this, the execution for CNN was closer, with Haskell trailing Python with less than double its time.

One of the reasons this could be is that Haskell does not have much support for its libraries, contrary to Python's machine learning and deep learning frameworks. Thus, the algorithms in the latter could have been tweaked to a larger extent to increase its performance.

Given this huge gap in performance, it is understandable to question the use of such a language. As we mentioned in the introduction, no language is the best, none is the worst, and Haskell has some areas where it excels.

Haskell's mathematical approach and its high level of abstraction promote problem solving at a high level. Since this is mainly applied through function application, the thought process learned from it, plus the particular implementations can usually be transferred to other languages.

The referential transparency assures that there is no possible mutable state and that some reference in a function will change its value. This leads to much safer and parallelizable code. If, for example, several tasks must be done simultaneously, by using Haskell, each function works independently, so there is no need to plan for the chance of different threads stepping on each other.

Haskell is a language that can be quickly prototyped, so it can prove useful when needing to implement new concepts in a short period of time. For example, Facebook implemented a spam detection program in Haskell [19] due to how easy rules are to implement, the fact that a new rule will not interfere with the previously implemented ones and that they can all run concurrently.

Other particular uses for the Haskell programming language:

- Suitable language for compilers.
- Easy code maintainability - due to its strong types, purity, global type inference, type classes and laziness.
- Type-driven development
- Single machine concurrency - runtime support for software-transactional memory.
- Parsing - using monads: `parsec`, `megaparse`, `attoparsec`

## 5.1. Future Work

This study may be extended in the future with several enhancements:

- To make full use of Haskell's innate ability to parallelize code, the performance of the tested algorithms could be compared with both languages running them on parallel instances.
- Using a more accurate performance profiler could lead to a better understanding of how each language allocates and manages memory, as well as to help provide other metrics which might be relevant.
- More machine learning algorithms could be tested, also for regression and not exclusively classification problems such as Support Vector Machines (SVMs) or k-nearest neighbors (k-NN).
- More datasets could be used in order to obtain more results and to understand in more detail how both the number of features and instances affects the training times for each of the algorithms.
- Finally, there could be accuracy tests implemented, to check whether the algorithms perform similarly in both languages, or one implementation has an odge over the other one.

# Bibliography


[1] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 778–788, IEEE, 2015.

[2] G. Van Rossum *et al.*, "Python programming language.," in *USENIX annual technical conference*, vol. 41, p. 36, 2007.

[3] S. Cass and B. Parthasaradhi, "Interactive: The top programming languages 2018," *IEEE Spectr.. https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages*, 2018.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[6] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, *et al.*, "Report on the programming language haskell: a non-strict, purely functional language version 1.2," *ACM SigPlan notices*, vol. 27, no. 5, pp. 1–164, 1992.

[7] M. Jaderberg, K. Simonyan, A. Zisserman, *et al.*, "Spatial transformer networks," in *Advances in neural information processing systems*, pp. 2017–2025, 2015.

[8] S. Tong and D. Koller, "Support vector machine active learning with applications to text classification," *Journal of machine learning research*, vol. 2, no. Nov, pp. 45–66, 2001.

[9] C. Taylor, "Haskell implementation of artificial intelligence algorithms [10]." https://github.com/chris-taylor/aima-haskell. Last accesed on 2019-06-13.

[10] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[11] H. Campbell, "grenade - deep learning in haskell." https://github.com/HuwCampbell/grenade. Last accesed on 2019-06-13.

[12] L. A. Breslow and D. W. Aha, "Simplifying decision trees: A survey," *The Knowledge Engineering Review*, vol. 12, no. 1, pp. 1–40, 1997.

[13] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[14] S. L. Salzberg, "C4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993," *Machine Learning*, vol. 16, pp. 235–240, Sep 1994.

[15] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1, pp. 278–282, IEEE, 1995.

[16] M. A. Nielsen, *Neural networks and deep learning*, vol. 25. Determination press San Francisco, CA, USA:, 2015.

[17] D. A. Van Dyk and X.-L. Meng, "The art of data augmentation," *Journal of Computational and Graphical Statistics*, vol. 10, no. 1, pp. 1–50, 2001.

[18] A. Cowley, "Frames: Data frames for working with tabular data files." http://hackage.haskell.org/package/Frames. Last accesed on 2019-06-13.

[19] S. Marlow, "Fighting spam with haskell," 2015.